

**Introducción**

**Introduccón**

En el presente trabajo se encomendó la tarea de desarrollar el diseño de un modelo para el desarrollo de una aplicación encargada en la detección y prevención de la enfermedad de Chagas. Dicha enfermedad es transmitida a humanos principalmente por un insecto llamado Vinchuca.

La tarea de la aplicación será la recopilación de muestras de insectos, la verificación de estas y el posterior aviso a las entidades interesadas en dichos eventos. Para esto contamos con diferentes entidades a modelar y sus relaciones.

**Modelaje de Ubicación**

Uno de los atributos más importantes del proyecto y que está presente a lo largo de buena parte de clases del modelo es el de Ubicacion. Las clases Muestra, ZonaDeCobertura y Organizacion, que veremos más adelante, poseen atributos que hacen alusión a esta clase y gran parte de sus cálculos se realizan a base de estos, por lo que dependen directamente.

Ante esta situación decidimos aplicar los principios SOLID y desarrollamos la interfaz Ubicación, que sabe responder su latitud, longitud y expresar la distancia en metros que hay hacia otra ubicación. Es esta interfaz con la que hemos tipado a las clases mencionadas arriba para que no dependan de una implementación concreta.

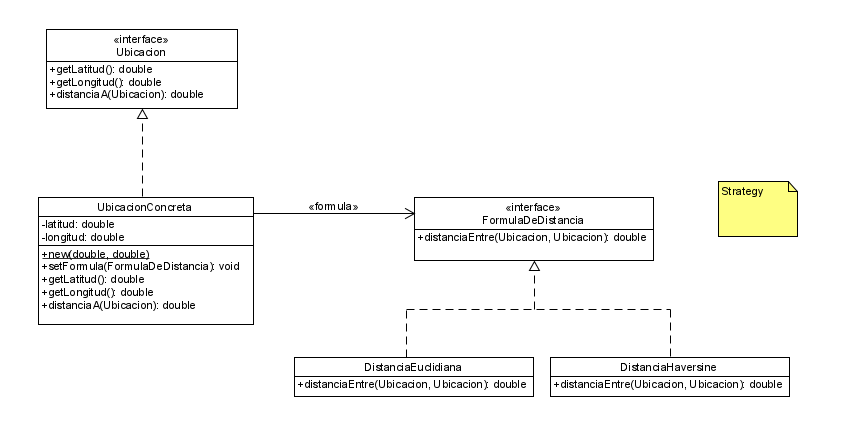
Implementando esta interfaz, desarrollamos la clase UbicacionConcreta, que responde correctamente el contrato de esta y además implementamos el **patrón Strategy**.

En un principio, la forma de calcular las distancias la hicimos acorde a la fórmula de Euclides para dos puntos en un plano XY. Más tarde encontramos que esta fórmula tiene algunas imprecisiones entre más cerca se encuentran los puntos a los polos de la tierra (ya que la tierra no es plana), por lo que encontramos que la fórmula adecuada para calcular la distancia entre dos puntos en el planeta es la fórmula Haversine.

Teníamos una sospecha de que esto podía ocurrir por lo que previamente habíamos modelado UbicacionConcreta para que tenga una variable de interfaz FormulaDeDistancia y que se pueda cambiar con un setter, por lo que modelar la fórmula Haversine se basó únicamente en crear esta clase, que implemente la interfaz y que sea el cálculo por defecto de UbicacionConcreta.

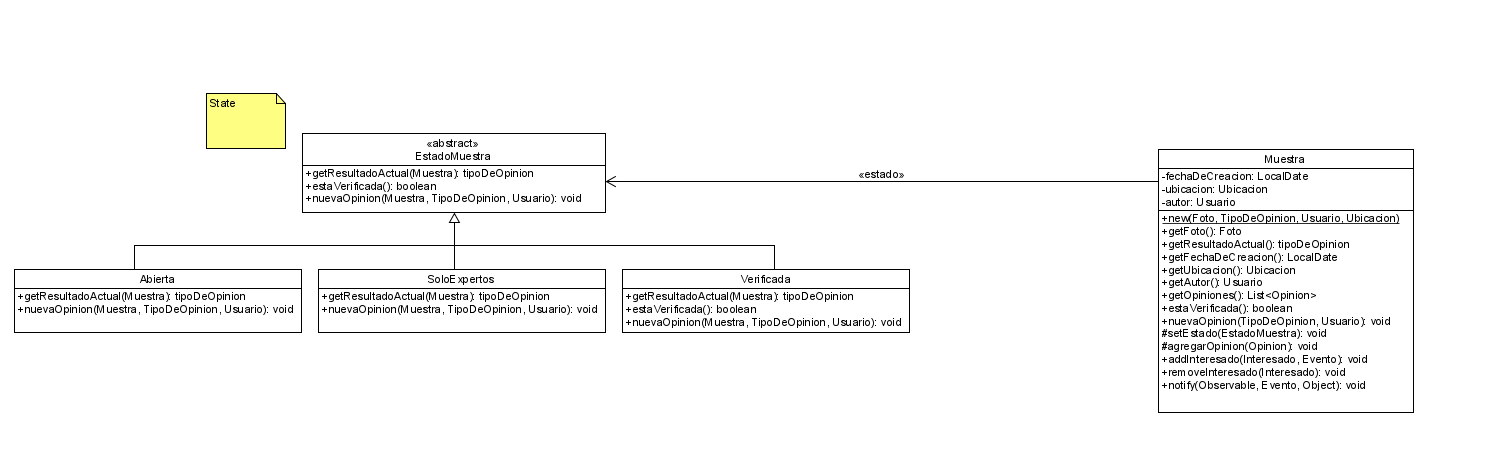
Los roles acorde al libro de Gamma serían:

* **Context:** UbicacionConcreta
* **Strategy:** FormulaDeDistancia
* **ConcreteStrategyA:** DistanciaEuclidiana
* **ConcreteStrategyB**: DistanciaHaversine



**Muestra**

Acorde al enunciado, la muestra es subida por un usuario, con el TipoDeOpinion que cree ser, a través de una aplicación móvil, y esta recibe opiniones de otros usuarios. La muestra debe saber responder cuál es el resultado actual (el TipoDeOpinion con mayor coincidencias entre las opiniones), y en caso de empate debe devolver TIPO\_NO\_DEFINIDO. El tipo de insecto lo representamos como un enumerativo con todos los casos posibles (lo cual es escalable si interesan agregar más). Así mismo creamos la clase Opinion que sabe responder si fue comentada por un experto, el autor y el TipoDeOpinion que es.  
 Las muestras cambian su comportamiento si un experto da su opinión en ellas. A partir de ese momento sólo pueden comentar expertos, y si dos coinciden con la opinión, la muestra pasa a estar verificada, no permitiendo a nadie más opinar. Esto es un claro **patrón State.** Por lo que desarrollamos las clases a continuación:



Donde los roles acorde a Gamma son:

* **Context:** Muestra
* **State:** EstadoMuestra
* **ConcreteStateA:** Abierta (permite comentar a todos)
* **ConcreteStateB:** SoloExpertos
* **ConcreteStateC:** Verificada (no deja comentar a nadie)

Además, planteamos y programamos el caso en que un usuario experto en el momento, suba una muestra. En dicha situación, la muestra no empezará en estado Abierta, sino en SoloExpertos.

**ZonaDeCobertura**

La implementación de dicha clase resultó ser simple. Posee una ubicación, una lista de muestras y sabe responder si solapa con otra zona y si una muestra está dentro de su radio de cobertura (en caso positivo, la añade a la lista si no estaba presente).

**Observer**

Detectamos un patrón observer en el modelo donde ZonaDeCobertura observa a Muestra, y Organizacion observa a ZonaDeCobertura. Notamos que este modelo es transitivo, por lo que en vísperas de lograr que no haya una dependencia transitiva, modelamos interfaces generales: Interesado y Observable. Estas interfaces tienen en los métodos notify y update poseen los parámetros Observable(la fuente), Evento (el evento que está interesado), Object (lo que disparó la notificación).

De esta forma logramos implementar con una interfaz el patrón para que funcione acorde a ZonaDeCobertura y Organizacion, que pretenden recibir objetos diferentes.

Cuando una muestra agrega una muestra a su lista, se suscribe a la muestra con el Evento VERIFICACION y notifica a la organización con el evento NUEVA\_MUESTRA. Finalmente cuando esta muestra se verifica, le notifica a Zona, la cual notifica a Organizacion.

Desarrollamos el modelo de esta forma con el objetivo mencionado anteriormente de evitar la dependencia transitiva y de poder ser escalable.

Los roles según Gamma:

* **Subject:** Observable
* **Observer:** Interesado
* **ConcreteSubject:** Muestra, ZonaDeCobertura
* **ConcreteObserver:** ZonaDeCobertura, Organizacion

**Usuario**

La clase Usuario encapsula la información de un usuario del sistema, incluyendo su nombre , las muestras que ha enviado y las opiniones realizadas. Además, implementa un sistema de clasificaciones por niveles de expertise (NivelExpertise, NivelBasico, NivelExperto, NivelExpertoExterno)que varían dinámicamente según la actividad reciente del usuario. Se ha pensado inicialmente como un Patrón State pero más tarde la idea fue descartada porque dichas clases no tienen comportamientos diferentes que las distinguen externamente, sin embargo dejamos representados los niveles de expertise por motivos de escalabilidad.  
  
Cada usuario posee un nivel que puede cambiar dinámicamente según su actividad reciente. El sistema evalúa, mediante el método verificarCambioNivel , si un usuario debe ascender o descender de categoría. Por ejemplo , un usuario con nivel básico se convierte en experto si ha enviado más de 10 muestras y emitido más de 20 opiniones en los ultimos 30 dias. En caso contrario, un experto puede volver a nivel básico si deja de cumplir esas condiciones. Esta condición se encapsula en un método común condicionExperto.

El nivel NivelExpertoExterno representa a un experto validado por afuera del sistema, y no puede cambiar de estado automáticamente, debido que esta validado por afuera que es experto.   
  
Debido a la implementación del patrón facilita la extensión del sistema , ya que en un futuro podremos agregar más nuevos niveles de y aplicar correctamente principio de diseño orientado a objetos como el Open/Closed y la única Responsabilidad.

**Filtros**

Se implementó un sistema de Filtros de muestras mediante el patrón Composite permitiendo combinar distintos criterios de filtrado de forma estructurada.  
  
La interfaz Filtro define un contrato común que deben cumplir todos los filtros: el método cumple(Muestra).

Se desarrollan filtros específicos como:

* FiltroTipoInsecto
* FiltroNivelVerificacion
* FiltroFechaDeCreacion
* FiltroUltimaOpinion

Estos filtros son hijos o hojas del patrón composite , es decir cumple el método cumple(Muestra) y terminan ahí su ejecución , no sigue agregando hijos (composite).

Mientras que los filtros:

* FiltroAND
* FiltroOR

Son Composite que agregan más filtros , cada uno con su comportamiento lógico.

Según Gamma los roles serían:

* **Component:** Filtro
* **Composite:** FiltroOR, FiltroAND
* **Leaf:** FiltroTipoInsecto, FiltroNivelVerificacion, FiltroFechaDeCreacion, FiltroUltimaOpinion

Esto permite construir filtros complejos reutilizando componentes simples.  
Además permite un bajo acoplamiento de dependencias a una clase única, permitiendo agregar más filtros sin modificar clases (Open/Closed) y que cada filtro tenga una única responsabilidad de decir si cumple esa función dependiendo que se quiere verificar en ese filtro.

**Organizacion**

La clase Organizacion, implementa el patrón Observer, permitiendo que las organizaciones reciban notificaciones automáticas cuando ocurre un evento en una zonaDeCobertura , como la carga de una nueva muestra o su verificación.

Cada organización reacciona a estos eventos a través del método update(), delegando la acción a una funcionalidad externa configurable ( FuncionalidadExterno). Esto permite definir distintos comportamientos para cada tipo de evento sin modificar la clase, logrando un diseño flexible y desacoplado.